Security Review Report
NM-0381 EtherFi CashSafe Top-Up Mechanism

NETHERMIND
SECURITY

(Nov 21, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the EtherFi Cash. It is a continuation based on the previous security reviews. This security engagement focused on a new feature that introduced a Top-Up mechanism that enables fast bridging of funds from directly to the `UserSafe` wallet. Other major changes were introduced in the `UserSafe` spending limits.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** eleven points of attention, where six are classified as `Low.` , and five are classified as `Informational.` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

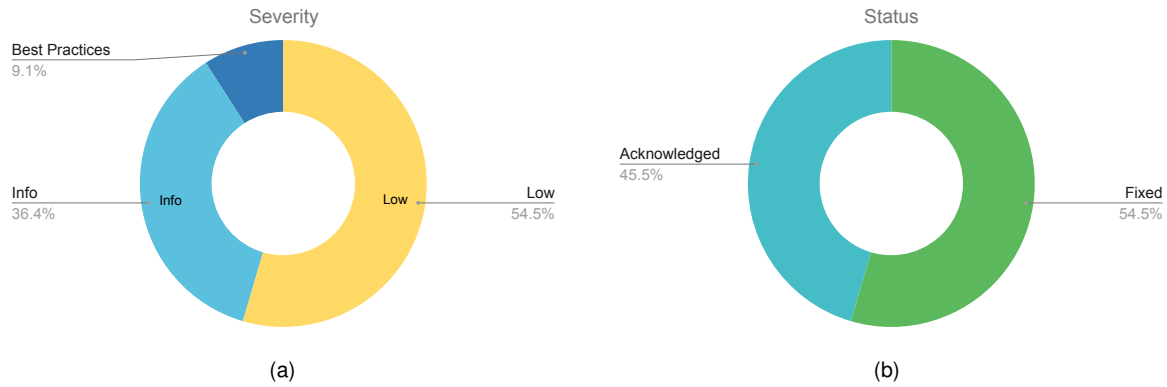

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (6), **Undetermined** (0), **Informational** (4), **Best Practices** (1). **Distribution of status: Fixed** (6), **Acknowledged** (5), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Nov 21, 2024
Response from Client	Regular responses during audit engagement
Final Report	Nov 22, 2024
Repository	cash-contracts
Commit (Audit)	95f3e06b729ac7726384b51ec6ba1248c50e9afe
Commit (Final)	2042b6b74c19297727e90b433fc6210611412ae7
Documentation Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	user-safe/UserSafeSetters.sol	232	3	1.3%	41	276
2	user-safe/UserSafeStorage.sol	89	14	15.7%	15	118
3	user-safe/UserSafeEventEmitter.sol	100	1	1.0%	27	128
4	user-safe/UserSafeCore.sol	357	18	5.0%	68	443
5	user-safe/UserSafeFactory.sol	77	11	14.3%	16	104
6	libraries/SignatureUtils.sol	33	4	12.1%	5	42
7	libraries/ArrayDeDupTransientLib.sol	23	7	30.4%	6	36
8	libraries/EIP1271SignatureUtils.sol	46	23	50.0%	9	78
9	libraries/UserSafeLib.sol	146	1	0.7%	18	165
10	libraries/TimeLib.sol	55	20	36.4%	20	95
11	libraries/SpendingLimitLib.sol	160	1	0.6%	34	195
12	settlement-dispatcher/SettlementDispatcher.sol	119	21	17.6%	27	167
13	top-up/TopUpSource.sol	149	8	5.4%	40	197
14	top-up/TopUpDest.sol	138	2	1.4%	30	170
15	top-up/bridges/EtherFiOFTBridgeAdapter.sol	56	2	3.6%	15	73
16	top-up/bridges/StargateAdapter.sol	63	3	4.8%	15	81
17	top-up/bridges/BridgeAdapterBase.sol	25	1	4.0%	7	33
	Total	**1868**	**140**	**7.5%**	**393**	**2401**

3 Summary of Issues

	Finding	Severity	Update
1	Admin can withdraw bridged user funds	Low	Acknowledged
2	Inconsistent behavior for the `block.timestamp = dailyRenewalTimestamp`	Low	Fixed
3	Lack of proper role distinction	Low	Acknowledged
4	Mapping addresses to wallets	Low	Acknowledged
5	No check for bridged amount size	Low	Acknowledged
6	Unnecessary updates of dailyRenewalTimestamp and monthlyRenewalTimestamp	Low	Fixed
7	Approval on `address(0)`	Info	Fixed
8	Missing input validation on timezone offset	Info	Fixed
9	No check to ensure the correctness of the token corresponding to the `StargateAdapter` contract	Info	Fixed
10	Non-standardised bridge implementation	Info	Acknowledged
11	Unused dummyLimit variables	Best Practices	Fixed

4 System Overview

The Top-Up mechanism is implemented with two contracts: `TopUpSource` and `TopUpDest`. Users can bridge funds from the source chain by transferring tokens to the `TopUpSource` contract. The off-chain infrastructure monitors the events emitted on transfer to that contract. Based on the information from the event, the off-chain infrastructure calls `TopUpDest.topUpUserSafe(...)` on the destination chain (Scroll) to transfer funds to the user's `UserSafe` wallet. The destination address of `UserSafe` should be registered by the user before the bridging. The funds on the `TopUpDest` must be predeposited by the EtherFi to allow initial bridging. The funds that are locked on the source chain can be bridged by the admin to the destination chain through LayerZero Stargate bridge, or OFT bridge. Below, we describe the most important public functions of the `TopUpSource` and `TopUpDest` contracts. `TopUpSource`:

- `bridge(...)` - transfers funds to the `TopUpDest` contract on the Sroll with LayerZero bridging mechanism. Restricted to bridger role.
- `recoverFunds(...)` - sends the funds back to the users that deposited tokens that can't be bridged. Restricted to default admin.

`TopUpDest`:

- `mapWalletToUserSafe(...)` - registers users address to `UserSafe`. The stored data must be signed by the address owner.
- `deposit(...)` - deposit funds that are used in the initial bridging process. Restricted to depositor role.
- `withdraw(...)` - withdraws the amount of funds equal to the previously deposited. Restricted to the default admin role.
- `topUpUserSafe(...)` - transfers funds from the `TopUpDest` to the defined `UserSafe` addresses.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Low] Admin can withdraw bridged user funds

File(s): `TopUpSource.sol`

Description: The function `recoverFunds(...)` allows the admin to withdraw any tokens from the `TopUpSource`. However, according to the documentation, this function should only withdraw tokens that are unsupported for bridging.

Recommendation(s): Consider restricting `recoverFunds(...)` function to allow only withdrawal the unsupported tokens.

Status: Acknowledged

6.2 [Low] Inconsistent behavior for the `block.timestamp = dailyRenewalTimestamp`

File(s): `SpendingLimitLib.sol`

Description: Consider the case when `block.timestamp = dailyRenewalTimestamp`.

If user has called `UserSafe::updateSpendingLimit` with `newDailyLimit >= limit.dailyLimit`, then the `SpendingLimitLib::updateSpendingLimit` function will set `dailyRenewalTimestamp` to the next day and `limit.spentToday` will not be reset so the user will have the `spentToday` not reset for an extra day.

```
function updateSpendingLimit(
    uint256 dailyLimitInUsd,
    uint256 monthlyLimitInUsd,
    bytes calldata signature
) external incrementNonce currentOwner {
    // ...

    (SpendingLimit memory oldLimit, SpendingLimit memory newLimit) = _spendingLimit.updateSpendingLimit(
        dailyLimitInUsd,
        monthlyLimitInUsd,
        _cashDataProvider.delay()
    );

    // ...
}

function updateSpendingLimit(
    SpendingLimit storage limit,
    uint256 newDailyLimit,
    uint256 newMonthlyLimit,
    uint64 delay
) external sanity(newDailyLimit, newMonthlyLimit) returns (SpendingLimit memory, SpendingLimit memory) {
    currentLimit(limit);
    SpendingLimit memory oldLimit = limit;

    if (newDailyLimit < limit.dailyLimit) {
        limit.newDailyLimit = newDailyLimit;
        limit.dailyLimitChangeActivationTime = uint64(block.timestamp) + delay;
    } else {
        limit.dailyLimit = newDailyLimit;
        // @audit Consider that block.timestamp is 00:00:00 of the current day, so the following line will set the limit
        ↪  to the next day
        limit.dailyRenewalTimestamp = uint256(block.timestamp).getStartOfNextDay(limit.timezoneOffset);
        limit.newDailyLimit = 0;
        limit.dailyLimitChangeActivationTime = 0;
    }

    // ...
}

function currentLimit(SpendingLimit storage limit) internal {
    SpendingLimit memory finalLimit = getCurrentLimit(limit);

    limit.dailyLimit = finalLimit.dailyLimit;
    limit.monthlyLimit = finalLimit.monthlyLimit;
    limit.spentToday = finalLimit.spentToday;
    limit.spentThisMonth = finalLimit.spentThisMonth;
    limit.newDailyLimit = finalLimit.newDailyLimit;
    limit.newMonthlyLimit = finalLimit.newMonthlyLimit;
    limit.dailyRenewalTimestamp = finalLimit.dailyRenewalTimestamp;
    limit.monthlyRenewalTimestamp = finalLimit.monthlyRenewalTimestamp;
    limit.dailyLimitChangeActivationTime = finalLimit.dailyLimitChangeActivationTime;
    limit.monthlyLimitChangeActivationTime = finalLimit.monthlyLimitChangeActivationTime;
    limit.timezoneOffset = finalLimit.timezoneOffset;
}

function getCurrentLimit(SpendingLimit memory limit) internal view returns (SpendingLimit memory) {
    // ...

    // @audit-q As block.timestamp == limit.dailyRenewalTimestamp, this if block will not be executed
    if (block.timestamp > limit.dailyRenewalTimestamp) {
        limit.spentToday = 0;
        limit.dailyRenewalTimestamp = getFinalDailyRenewalTimestamp(limit.dailyRenewalTimestamp, limit.timezoneOffset);
    }

    // ...
}
```

The same is the case with `monthlyRenewalTimestamp`.

Recommendation(s): Consider updating the `limit.dailyRenewalTimestamp` only in the `getCurrentLimit(...)` function.

Status: Fixed

Update from the client: Fixed in 3135867264611c08f581da6938e852e26f0e979c.

6.3 [Low] Lack of proper role distinction

File(s): `TopUpSource.sol`, `TopUpDest.sol`

Description: The roles are not clearly sperated. The `DEFAULT_ADMIN_ROLE` is able to do multiple critical actions like withdrawals or recovering funds.

Recommendation(s): Consider clearly specifying sparate roles for critical functions.

Status: Acknowledged

6.4 [Low] Mapping addresses to wallets

File(s): `TopUpSource.sol`, `TopUpDest.sol`

Description: To bridge funds from the source chain to the Scroll, users are expected to first assign their address to the `UserSafe` wallet by calling `TopUpDest.mapWalletToUserSafe(...)`. During the bridging process, the off-chain infrastructure fetches registered addresses from the `walletToUserSafeRegistry` and calls `TopUpDest.topUpUserSafe(...)` to transfer funds to the users `UserSafes`. However, if users would perform the bridging without registering the address on `TopUpDest`, the funds would be locked in `TopUpSource`. Currently, the funds can be sent back to the user with `recoverFunds(...)`, but this function according to the specification, should only transfer unsupported tokens.

Recommendation(s): Consider adding a mechanism that would allow users to get the funds back on bridging without previous registration.

Status: Acknowledged

6.5 [Low] No check for bridged amount size

File(s): `TopUpSource.sol`

Description: The current bridging does not validate the amounts requested by the users to bridge. If a user sends a high amount of funds, the `TopUpDest` contract may not have enough funds to release on the destination chain. This would require the EtherFi team to add more funds to the `TopUpSource` or bridge funds from the source chain, which would force the user to wait. Another possible scenario is creating many requests for small or zero amounts that would trigger the bridging process and potentially slow down or temporarily DOS the bridge due to the high volume of transfers.

Recommendation(s): Since, with the current design, there is no way to check the number of incoming tokens, consider handling the potential problems with the amount on the off-chain side.

Status: Acknowledged

6.6 [Low] Unnecessary updates of dailyRenewalTimestamp and monthlyRenewalTimestamp

File(s): `SpendingLimitLib.sol`

Description: The renewal timestamps are currently updated in two cases: i) when the renewal period passes or ii) when the spending limit is changed. However, the update in the second case is unnecessary or may even lead to unwanted behavior under certain conditions. Let's first consider the update of the spending limit with `newDailyLimit >= limit.dailyLimit` (note that all examples are equally applied to monthly limits). Case A.1 1) The User calls `updateSpendingLimit(...)` with `newDailyLimit >= limit.dailyLimit`, the limit is updated 2) Assume that the `block.timestamp > limit.dailyRenewalTimestamp`. In the `getCurrentLimit(...)`, the spending is reset, and the `limit.dailyRenewalTimestamp` is updated to the start of the next day at `timestamp = 0am Nov 9` 3) Then `updateSpendingLimit(...)` executes further and enters the `else` branch, where the `dailyRenewalTimestamp` is updated again, but results in the same `timestamp = 0am Nov 9`:

```
else {
  limit.dailyLimit = newDailyLimit;
  // @audit: assume the current timestamp is 1pm Nov 8, then the start of the next day will be 0am Nov 9
  limit.dailyRenewalTimestamp = uint256(block.timestamp).getStartOfNextDay(limit.timezoneOffset);
  limit.newDailyLimit = 0;
  limit.dailyLimitChangeActivationTime = 0;
}
```

Case A.2 1) The user calls `updateSpendingLimit(...)` with `newDailyLimit >= limit.dailyLimit`, the limit is updated 2) Assume that the `block.timestamp <= limit.dailyRenewalTimestamp`. Nothing happens in the `getCurrentLimit(...)` 3) In `updateSpendingLimit(...)`, the `limit.dailyRenewalTimestamp` is set to the same value as it was

```
// @audit: since the current timestamp is lower than the renewal timestamp, this calculation resolves to the already
 ↪  stored renewal timestamp
limit.dailyRenewalTimestamp = uint256(block.timestamp).getStartOfNextDay(limit.timezoneOffset);
```

As can be seen, the renewal timestamp update in the `updateSpendingLimit(...)` when increasing the limit is unnecessary. In the case when the limit is decreased, there is a delay. Let's consider now this scenario: Case B.0 1) The user calls `updateSpendingLimit(...)` with `newDailyLimit < limit.dailyLimit`. `newDailyLimit` and `dailyLimitChangeActivationTime` are set 2) The user spends funds and enters the first branch in `getCurrentLimit(...)` since the current timestamp is greater than `limit.dailyLimitChangeActivationTime`. The `limit.dailyLimit` and `limit.dailyRenewalTimestamp` are updated. Now, there are two possibilities:

Case B.1

- The `dailyRenewalTimestamp` is lower than the current timestamp, and we enter the block (in `getCurrentLimit(...)`):;

```
if (block.timestamp > limit.dailyRenewalTimestamp) {
    limit.spentToday = 0;
    limit.dailyRenewalTimestamp = getFinalDailyRenewalTimestamp(limit.dailyRenewalTimestamp,
    ↪  limit.timezoneOffset);
}
```

In this block the `dailyRenewalTimestamp` is updated to a new value, beginning of the next day. The update of renewal timestamp in case B.0 is then redundant.

Case B.2

- The `dailyRenewalTimestamp` is equal to or greater than the current timestamp, and the `limit.spentToday` is not reset. This is an issue since the renewal timestamp was updated (in Case B, step 2)) but the spending was not reset. That means the user would need to wait another day or month to get the spending reset;

Recommendation(s): The update of the renewal timestamp should only be done in the following statements in `getCurrentLimit(...)`:

```
if (block.timestamp > limit.dailyRenewalTimestamp) {
    limit.spentToday = 0;
    limit.dailyRenewalTimestamp = getFinalDailyRenewalTimestamp(limit.dailyRenewalTimestamp, limit.timezoneOffset);
}

if (block.timestamp > limit.monthlyRenewalTimestamp) {
    limit.spentThisMonth = 0;
    limit.monthlyRenewalTimestamp = getFinalMonthlyRenewalTimestamp(limit.monthlyRenewalTimestamp,
    ↪  limit.timezoneOffset);
}
```

This way redundant updates presented in cases A.1, A.2 and B.0 could be avoided. The case B.2 with incorrect extension of the renewal period could also be resolved by removing the renewal timestamp update from the first two `if` blocks in `getCurrentLimit(...)`.

Status: Fixed

Update from the client: Fixed in 91c59af84255188088b360091ed5f196683bac0c and 3135867264611c08f581da6938e852e26f0e979c.

6.7 [Info] Approval on `address(0)`

File(s): `StargateAdapter.sol`, `SettlementDispatcher.sol`

Description: The `bridge(...)` function calls `approve` on the token to bridge, but in the case of a native token, it passes `value` while calling the `IStargate(stargatePool).sendToken(...)` function. But even in the case of the native token, `approve` is called, which will revert the transaction as the native token is represented by a zero address, which would have `code.length == 0`. Although, as per the current implementation, zero address, i.e., native token, won't be passed in the `bridge(...)` function, it is considered in `prepareRideBus(...)` by checking `if (IStargate(stargate).token() == address(0))`, so to be consistent and make the code future proof, it should be ensured that `approve` is only called if `token` isn't zero address i.e. native token.

```solidity
    function bridge(
        address token,
        uint256 amount,
        address destRecipient,
        uint256 maxSlippage,
        bytes calldata additionalData
    ) external payable override {
        // ...

        (uint256 valueToSend, SendParam memory sendParam, MessagingFee memory messagingFee) =
            prepareRideBus(stargatePool, amount, destRecipient, minAmount);

        if (address(this).balance < valueToSend) revert InsufficientNativeFee();

        // @audit It will fail if `token` is a zero address i.e., a native token
        IERC20(token).forceApprove(stargatePool, amount);

        // @audit In the case of the native token, the amount is transferred as `value`
        (, , Ticket memory ticket) = IStargate(stargatePool).sendToken{ value: valueToSend }(sendParam, messagingFee,
         ↪  payable(address(this)));
        emit BridgeViaStargate(token, amount, ticket);
    }

    // from
     ↪  https://stargateprotocol.gitbook.io/stargate/v/v2-developer-docs/integrate-with-stargate/how-to-swap#ride-the-bus
    function prepareRideBus(
        address stargate,
        uint256 amount,
        address destRecipient,
        uint256 minAmount
    ) public view returns (uint256 valueToSend, SendParam memory sendParam, MessagingFee memory messagingFee) {
        // ...

        if (IStargate(stargate).token() == address(0)) {
            // @audit In case of native tokens, the amount would be passed as `value` while calling the `sendToken`
             ↪  function
            valueToSend += sendParam.amountLD;
        }
    }
```

Recommendation(s): Add a check in the `bridge(...)` function to ensure that `approve(...)` is called only if the `token` passed isn't a zero address, i.e., native token.

Status: Fixed

Update from the client: Fixed in cd12ca0edc70cb66f9f23ae18b2f92ceee4ed64d.

6.8 [Info] Missing input validation on timezone offset

File(s): `SpendingLimitLib.sol`

Description: The `timezoneOffset` is not bound to a reasonable range of `24` hours. This potentially allows for incorrect time zones to be set.

Recommendation(s): Consider bounding the `timezoneOffset` to `24` offset.

Status: Fixed

Update from the client: Fixed in 91c59af84255188088b360091ed5f196683bac0c.

6.9 [Info] No check to ensure the correctness of the token corresponding to the `StargateAdapter` contract

File(s): `StargateAdapter.sol`, `SettlementDispatcher.sol`

Description: The `bridge(...)` function takes `token` as an argument and `additionalData`, which is decoded into the `stargatePool` address. There isn't any check that ensures that the token corresponding to the `stargatePool` contract is the same as the `token` passed in the `bridge(...)` function. This can be an issue, especially if the `stargatePool` contract corresponds to the native token, whereas the `token` passed in the `bridge(...)` function doesn't correspond to the native token (let's say erc20 token X). In that case, as native tokens are passed as `value` while calling `IStargate(stargatePool).sendToken(...)` function, the transaction would not fail if sufficient `value` is passed while calling `bridge(...)` function. The `bridge(...)` function will call `approve(...)` in the token X contract, whereas the token transferred will be a native token.

```
    function prepareRideBus(
        address stargate,
        uint256 amount,
        address destRecipient,
        uint256 minAmount
    ) public view returns (uint256 valueToSend, SendParam memory sendParam, MessagingFee memory messagingFee) {
        // ...

        if (IStargate(stargate).token() == address(0)) {
            // @audit In case of native tokens, the amount would be passed as `value` while calling the `sendToken`
             ↪  function
            valueToSend += sendParam.amountLD;
        }
    }

    function bridge(
        address token,
        uint256 amount,
        address destRecipient,
        uint256 maxSlippage,
        bytes calldata additionalData
    ) external payable override {
        // ...

        (uint256 valueToSend, SendParam memory sendParam, MessagingFee memory messagingFee) =
            prepareRideBus(stargatePool, amount, destRecipient, minAmount);

        if (address(this).balance < valueToSend) revert InsufficientNativeFee();

        // @audit calling `approve(...)` on token passed in `bridge(...)` function
        IERC20(token).forceApprove(stargatePool, amount);

        // @audit If sufficient `value` is passed while calling this function, then in case of native token, it will
         ↪  succeed.
        (, , Ticket memory ticket) = IStargate(stargatePool).sendToken{ value: valueToSend }(sendParam, messagingFee,
         ↪  payable(address(this)));
        emit BridgeViaStargate(token, amount, ticket);
    }
```

Recommendation(s): Add a check in the `bridge(...)` function to ensure that the `token` passed as an argument is the same as the `token` corresponding to `stargatePool`. It can be done by adding this line in the `bridge(...)` function:

```
    error IncorrectStargatePool();

    function bridge(
        address token,
        uint256 amount,
        address destRecipient,
        uint256 maxSlippage,
        bytes calldata additionalData
    ) external payable override {
        // ...

        if (address(token) != address(IStargate(stargatePool).token())) revert IncorrectStargatePool();

        // ...
    }
```

Status: Fixed

Update from the client: Fixed in cd12ca0edc70cb66f9f23ae18b2f92ceee4ed64d.

6.10 [Info] Non-standardised bridge implementation

File(s): .

Description: The current implementation of the bridging process is based on EtherFi's off-chain infrastructure that reads on-chain events to trigger transfers on the destination chain. The off-chain part is out of scope, but during the security review, there was no specification for the bridge. The implementations of the bridges historically faced many problems and exploits since the creation of a secure bridge is a difficult task. Therefore, we strongly recommend performing security reviews of the design and implementation of the off-chain part that is responsible for the bridging.

Recommendation(s): Secure the off-chain part of the bridge before deployment. Additionaly, consider open-surcing the code to allow users to review.

Status: Acknowledged

6.11 [Best Practices] Unused dummyLimit variables

File(s): `SpendingLimitLib.sol`, `UserSafeCore.sol`

Description: The functions `SpendingLimitLib.initialize` and `UserSafeCore.emitInitializeEvents` have a variable `dummyLimit` which is declared and emitted or returned, but provides no useful information. For the `initialize` function, the returned data is then unused by the calling function `UserSafeCore.initialize`. The functions are shown below:

```
// SpendingLimitLib
function initialize(...) external ... returns (SpendingLimit memory, SpendingLimit memory) {
    SpendingLimit memory dummyLimit;
    //...
    return (dummyLimit, limit);
}

// UserSafeCore
function initialize(...) external initializer {
    //...
    // The `dummyLimit` returned data is not used
    (, SpendingLimit memory newLimit) = _spendingLimit.initialize(
        __dailySpendingLimit,
        __monthlySpendingLimit,
        __timezoneOffset
    );
    //...
}
```

Recommendation(s): Consider removing the `dummyLimit` variables from the `SpendingLimitLib.initialize` and `UserSafeCore.emitInitializeEvent` functions.

Status: Fixed

Update from the client: Fixed in 91c59af84255188088b360091ed5f196683bac0c.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the EtherFi Cash documentation

The documentation for the EtherFi Cash protocol was provided by the team during kick-off meeting.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project.

8 Test Suite Evaluation At Final Commit

```
% forge test
[] Compiling...
[] Compiling 241 files with 0.8.24
[] Solc 0.8.24 finished in 56.32s

Ran 32 test suites in 6.76s (24.42s CPU time): 259 tests passed, 0 failed, 0 skipped (259 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer